CSCI 620 – Intro to Big Data

Assignment #2 – Relational Model

Name: *Yuvraj Singh*

1a) Creating the Database

The database was created by running PSQL and connecting to Postgres. After connecting, the database was created with the command *"CREATE DATABASE assignment2;"*. After the database is created, the tables were created using Windows PowerShell.

1b) Preparing the Data

The initial data files were preprocessed and cleaned using Python. To clean the data files, make sure that each data file is in its own respective folder in the working directory and then run *clean_tsvs.py*. The only dependency that will need to be installed is *Pandas*. The python script will clean the data files needed for this assignment and will create new csv files in the data folder. The script will also output the time it took to clean the files and any errors that was logged. The overall time it took to clean the data was approximately 10.5 minutes.

```
It took 115.0243794 seconds to clean and parse out title information
It took 51.217196900000005 seconds to clean and parse out members
It took 448.83386590000003 seconds to clean and parse out member informations and relationships

(bigdata) C:\Users\Yuvraj\Desktop\Fall_2020\CSCI-620\assignments\assignment2>
```

1c) Populating the Database

Each table in the database is populated using Python and the copy command for Postgres. To populate the tables, run *pop_db.py*. The only dependency that this script will need is *Psycopg2* which is the python library for operating a Postgres database. For the basic tables such as title, member, genres, etc.… the script will perform a Postgres copy command to load the csv data straight into its respective

table. As for the relationship tables, the script will create a temporary table to load the csv data in and will then insert each distinct row of the temporary table into the relationship tables. The overall time it took to populate the database was approximately 23 minutes.

```
(bigdata) C:\Users\Yuvraj\Desktop\Fall_2020\CSCI-620\assignments\assignment2> cd c:\Users
r 51546 -- c:\Users\Yuvraj\Desktop\Fall_2020\CSCI-620\assignments\assignment2\pop_db.py "
It took 51.834696300000004 seconds to create the basic tables
It took 1330.9533877 seconds to create the basic tables

(bigdata) C:\Users\Yuvraj\Desktop\Fall_2020\CSCI-620\assignments\assignment2>
```

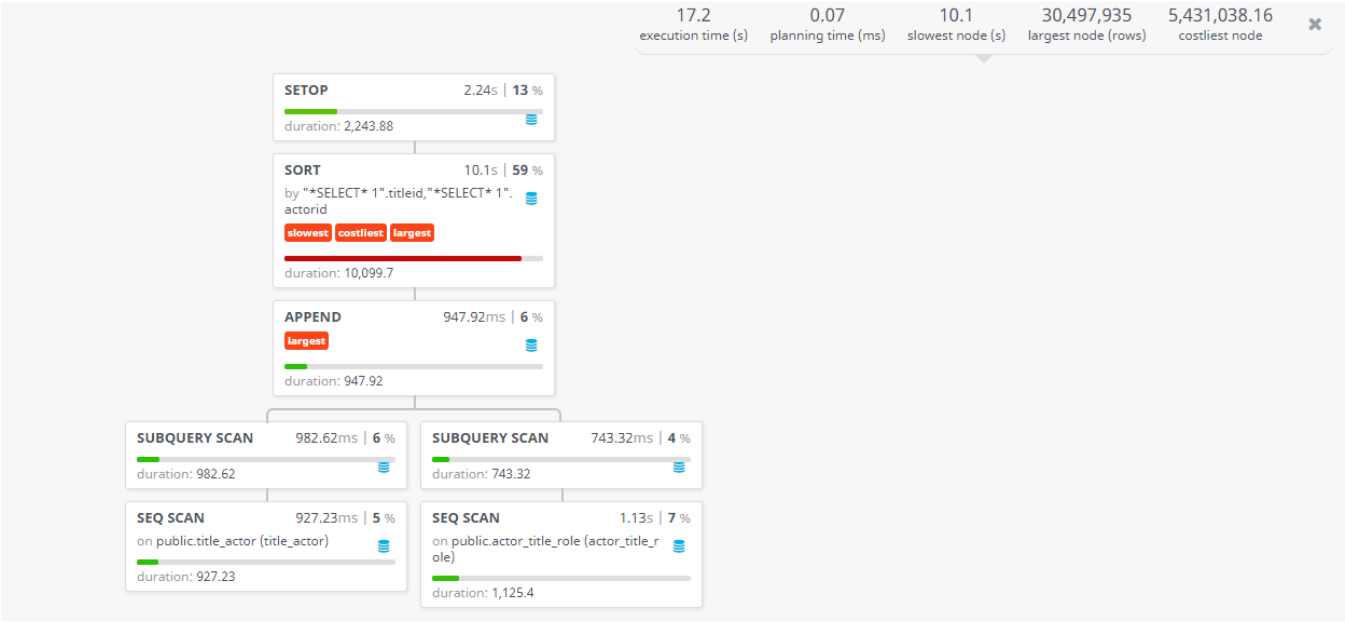
2) Custom Queries

The results for the custom queries for question two are provided in the *pop_db.py* script. To obtain the results, uncomment the *db.queries()* function call in main method and run the script. As each query is performed, a small sample set of the result is printed out along with the amount of time it took to perform the query. The time printed out will include the small amount of time needed to perform the correct function calls in the script so it will not be 100% exact but it will be close to the exact time needed to perform the query.

3) Custom Query Visualization

For each custom query, a JSON file is created that contains the Postgres EXPLAIN information for each query. Each query is then visualized using http://tatiyants.com/pev/ which will display the io costs and the duration for each step.

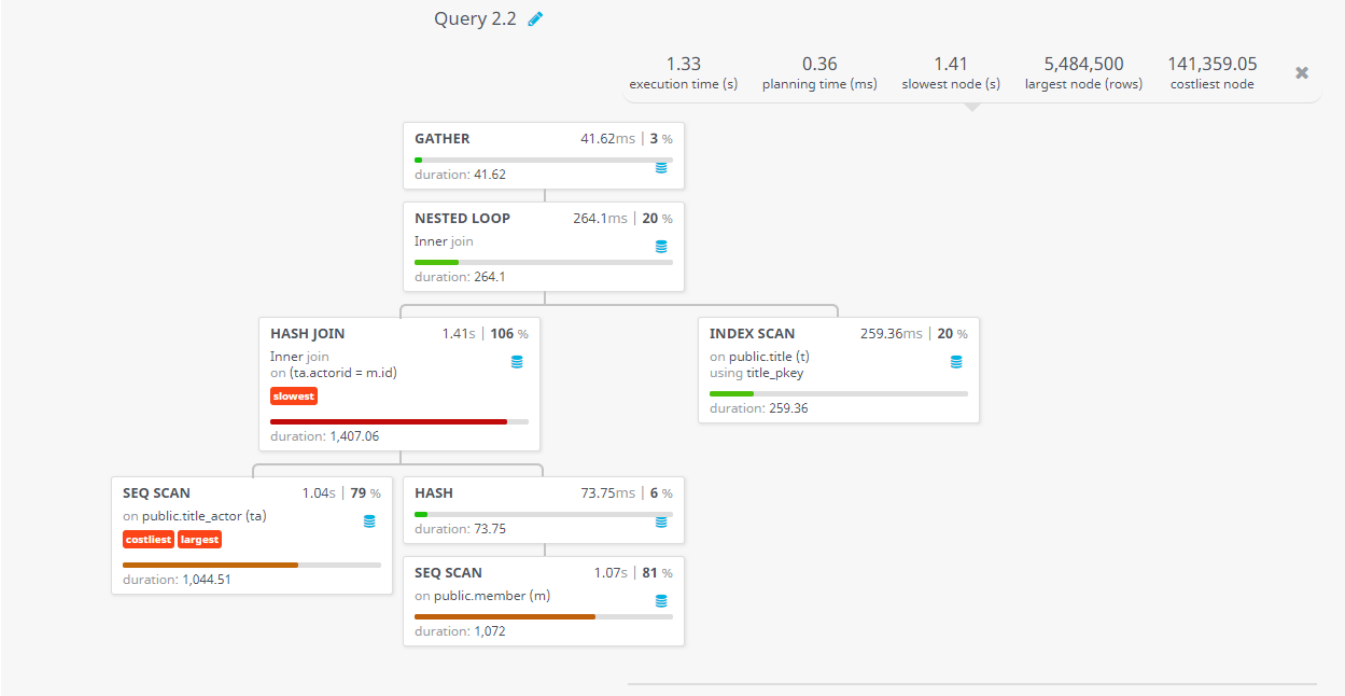
Query 2.1



With the first custom query, we can see the steps required to perform the query and how long certain steps took with additional information. As you can see, the first steps that occurred were sequential scans which is the most basic way to fetch rows from a table. A sequential scan is just iterating through a table, one row at a time and returning the rows requested in the query. A sequential scan is performed on both title_actor and actor_title_role. After the sequential scans are finished, a subquery scan is then performed on each sequential scan, which is simply scanning the output of a sub-query and filtering.

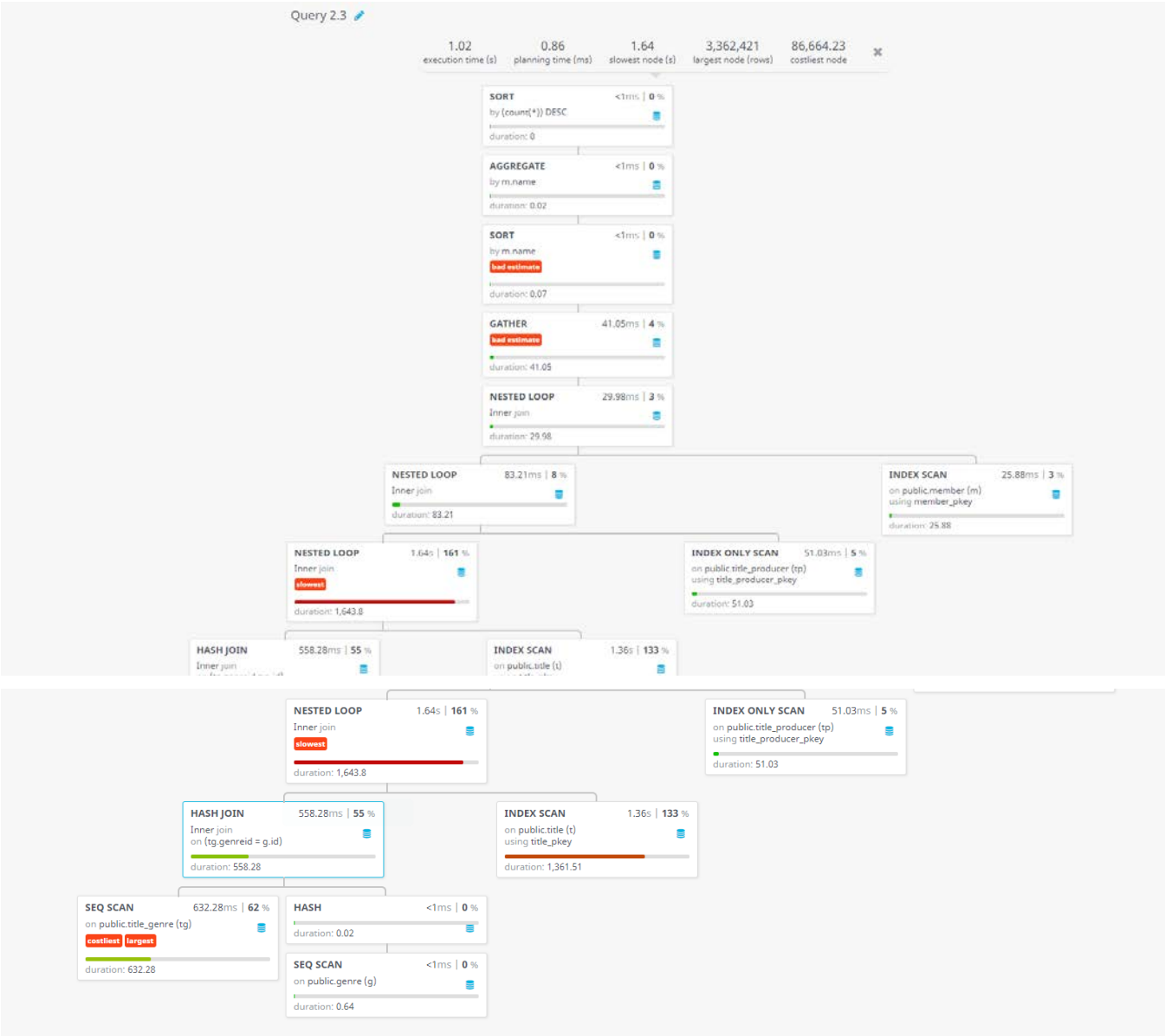
Next, the two scans are then appended together and then sorted. The sorting block it what takes the most time in the query and it is also the largest in size and cost.

Query 2.2



With this custom query, two sequential scans are started on the title_actor table and the member table. Since the purpose of this query is to return some members who are identified as actors and have not played in movies in 2014, it would make sense for these two seq scans to take quite a bit of time. After the sequential scan on the member table is finished, it is then hashed so the two sequential scans can be hash joined on the member id and title_actor member foreign key. An index scan of the title table then occurs to find movies not in 2014. After the hash join and index scan are done, they are then inner joined with a nested loop.

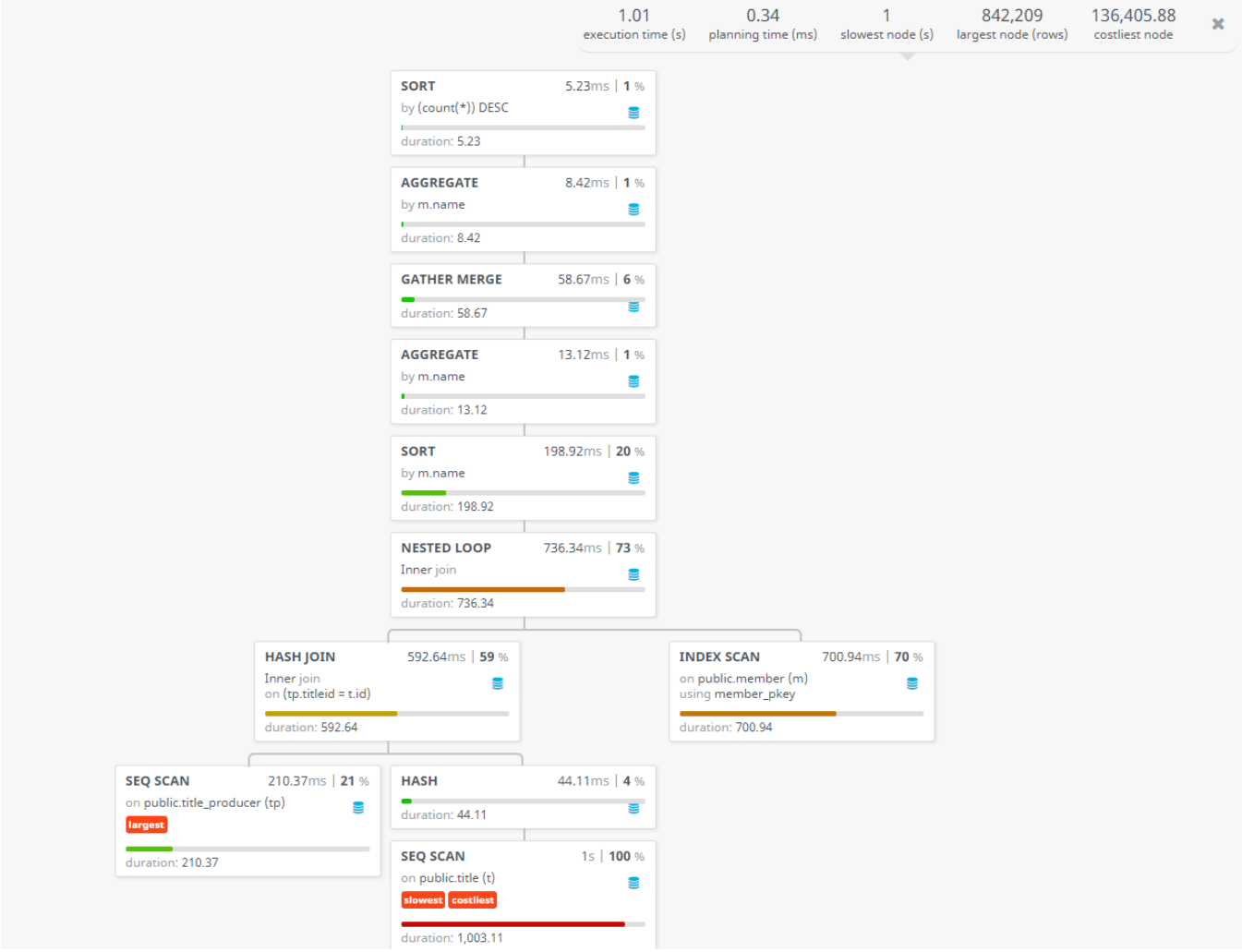
Query 2.3



The third query has a lot of steps due to the complexity of the query. The query itself needs data from about five tables which does add some time to the query execution time. The first things to begin are two sequential scans on the title_genre table and the genre_table. The sequential scan of the genre table is then hashed so the two scans can be hash joined on genre ids. While the hash join is operating, an index scan on the title table then begins. Afterwards, the result of the initial hash join, and the index scan are

then joined with a nested loop. An index only scan on the title_producer table then starts as well. The results from the nested loop and index only scan are then joined together with another nested loop who is then furthered joined in a nested loop with an index scan on the member table. After all the joins, the results are then gathered and sorted for the desired output.

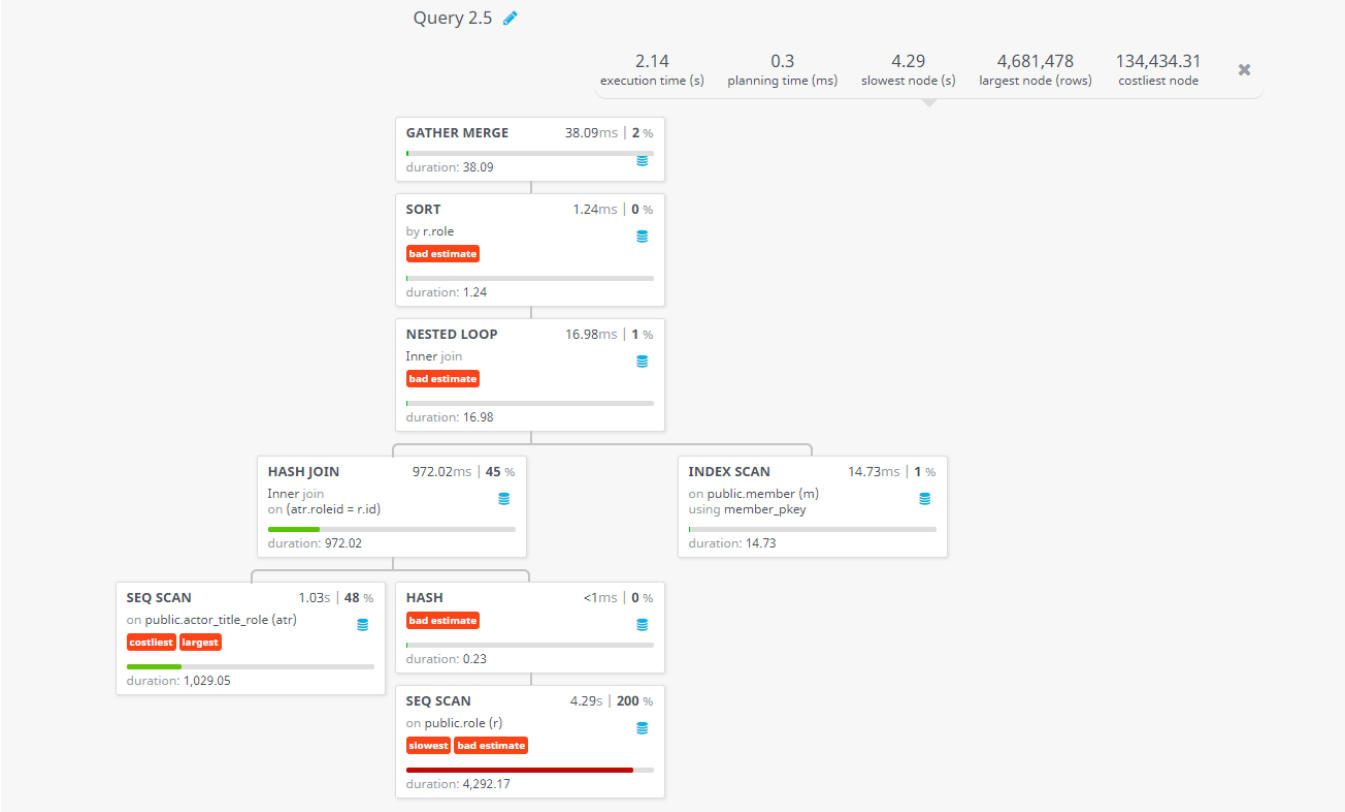
Query 2.4



The fourth query starts off with a sequential scan on the title table. A sequential scan on the title_poducer table is then started and with the first sequential scan being hashed so a hash join can be

performed on both scans. An index scan on the member table is then started and is joined with the results from the hash join using a nested loop. After the nested loop is done, the results are then sorted by member.name, aggregated, merged, aggregated and sorted one last time. The initial sequential scan on the title table is what cost the most and took the longest.

Query 2.5



For the last query, a sequential scan is started on the role table and then hashed. A second sequential scan is started on the actor_title_role and the two are then hash joined on the role id. An index scan on the member table then begins and is then joined with the results from the hash join using a nested loop. The results from the nested loop is then sorted and merged. The sequential scan of the role table is what took the longest time to perform while the sequential scan on the actor_title_role cost the most.

4) Relational Algebra for Custom Queries

(Q 2.1) – $\pi_{\text{titleid, actorid}}$ (title_actor – actor_title_role)

(Q 2.2) – $\sigma_{\text{(name like 'Phi%', deathYear = NULL)}}$ ($\sigma_{\text{(type = 'movie' and startYear != 2014 and endYear != 2014)}}$($\sigma_{\text{(member.id = title_actor.actorid)}}$(member x $\sigma_{\text{(title_actor.titleid = title.id)}}$(title_actor x title))))

(Q 2.3) – $\pi_{\text{(member.name)}}$($\sigma_{\text{(name like '%Gill%')}}$ ($\sigma_{\text{(startYear = 2017 or endYear = 2017)}}$($\sigma_{\text{(genre = 'Talk-Show)}}$ ($\sigma_{\text{(member.id = title_producer.producerid)}}$ (member x $\sigma_{\text{(title_producer_titleid = title.id)}}$(title_producer x $\sigma_{\text{(title.id = title_genre.titleid)}}$ (title x $\sigma_{\text{(title_genre.genreid = genre.id)}}$ (title_genre x genre))))))

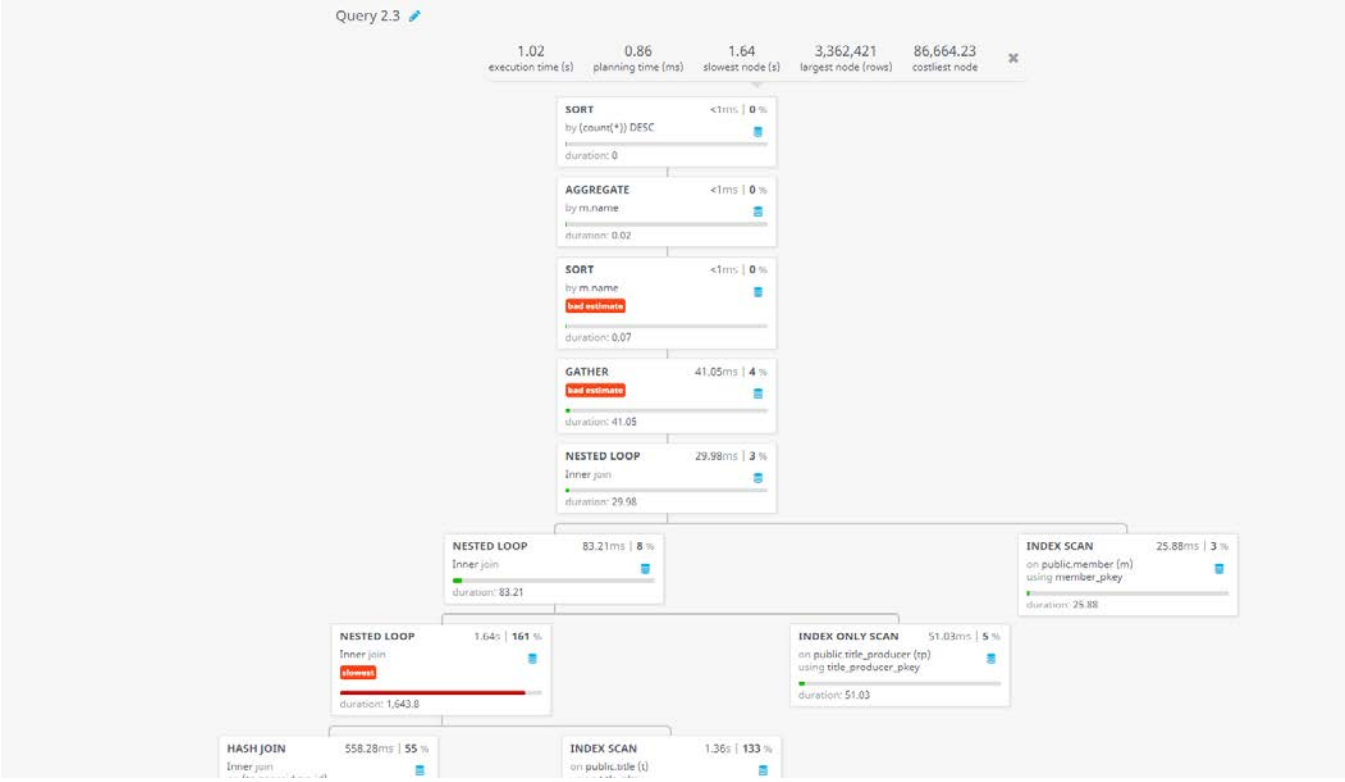
(Q 2.4) – $\pi_{\text{(member.name)}}$ ($\sigma_{\text{(deathYear = NULL)}}$($\sigma_{\text{(runtime > 120)}}$ ($\sigma_{\text{(member.id = title_producer.producerid)}}$ (member x $\sigma_{\text{(title_producer_titleid = title.id)}}$(title_producer x title))))

(Q 2.5) – $\pi_{\text{(member.name, role.role)}}$ ($\sigma_{\text{(deathYear = NULL)}}$($\sigma_{\text{(member.id =actor_title_role.actorid)}}$ (member x $\sigma_{\text{(actor_title_role.roleid = role.roleid)}}$(actor_title_role x role))))

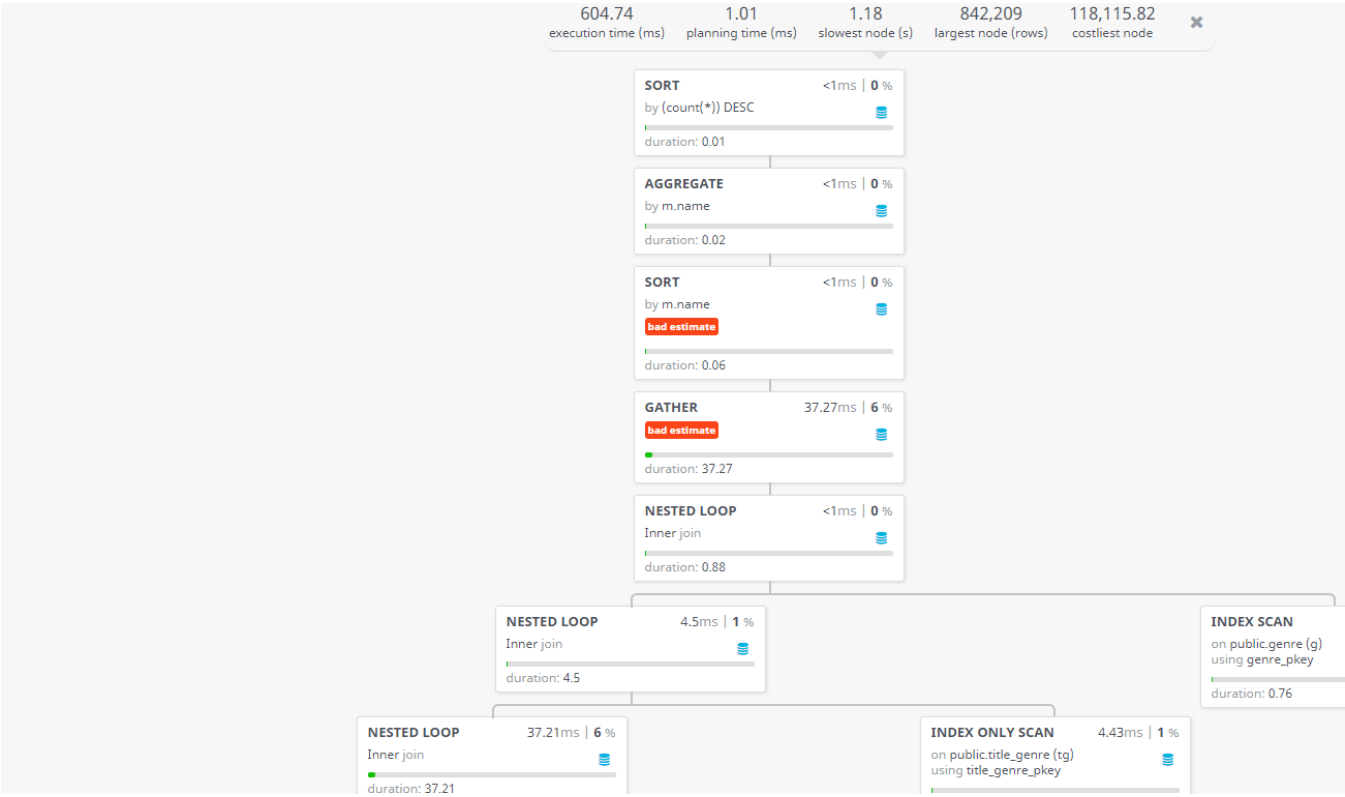
5) Creating Indexes

Indexes were created for most of the queries to help with performance and execution timing. Many of the indexes were chosen based on the filtering clauses for each query. In the *pop_db.py* script, a function is defined to create the indexes for certain tables based what was deemed helpful for each set of queries. For example, the third query had an execution time of approximately one second. After adding indexes for member.names, title.startYear, and title.endYear, the execution time was cut down to approximately 600 milliseconds. Below is a before and after indexes picture for the query information.

Before:



After:



The indexing did help some of the queries, but for some it did not make much of a difference. For the third query, a bitmap index scan was used in the beginning instead of a sequential scan on the title table and this made quite a big difference in performance and timing, but it caused the hash join to take a bit longer. For this query, indexes were added to the title.runtime and member.deathYear along with all of the previous indexes, and because of everything, a 20%-30% increase in performance is seen.